Proceedings of the Linux Symposium

Volume One

July 20nd–23th, 2005
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
C. Craig Ross, *Linux Symposium*
Stephanie Donovan, *Linux Symposium*

Review Committee

Gerrit Huizenga, *IBM*
Matthew Wilcox, *HP*
Dirk Hohndel, *Intel*
Val Henson, *Sun Microsystems*
Jamal Hadi Salimi, *Znyx*
Matt Domsch, *Dell*
Andrew Hutton, *Steamballoon, Inc.*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*

Block Devices and Transport Classes: Where are we going?

James E.J. Bottomley
SteelEye Technology, Inc.
jejb@steeleye.com

Abstract

A transport class is quite simply a device driver helper library with an associated sysfs component. Although this sounds deceptively simple, in practise it allows fairly large simplifications in device driver code. Up until recently, transport classes were restricted to be SCSI only but now they can be made to apply to any device driver at all (including ones with no actual transports).

Subsystems that drive sets of different devices derive the most utility from transport classes. SCSI is a really good example of this: We have a core set of APIs which are needed by every SCSI driver (whether Parallel SCSI, Fibre Channel or something even more exotic) to do command queueing and interpret status codes. However, there were a large number of ancillary services which don't apply to the whole of SCSI, like Domain Validation for Parallel SCSI or target disconnection/reconnection for Fibre Channel. Exposing parameters (like period and offset, for parallel SCSI) via `sysfs` gives the user a well known way to control them without having to develop a core SCSI API. Since a transport class has only a `sysfs` interface and a driver API it is completely independent of the SCSI core. This makes the classes arbitrarily extensible and imposes no limit on how many may be simultaneously present.

This paper will examine the evolution of the transport class in SCSI, covering its current uses in Parallel SCSI (SPI), Fibre Channel (FC) and other transports (iSCSI and SAS), contrasting it with previous approaches, like CAM, and follow with a description of how the concept was freed from the SCSI subsystem and how it could be applied in other aspects of kernel development, particularly block devices.

1 Introduction

Back in 1986, when the T10 committee first came out with the Small Computer Systems Interconnect (SCSI) protocol, it was designed to run on a single 8 bit parallel bus. A later protocol revision: SCSI-2 [1] was released in 1993 which added the ability to double the bus width and do synchronous data transfers at speeds up to 10MHz. Finally, in 1995, the next generation SCSI-3 architecture [5] was introduced. This latest standard is a constantly evolving system which includes different transports (like serial attached SCSI and Fibre Channel) and enhances the existing parallel SCSI infrastructure up to Ultra360.

2 Overview of SCSI

From its earliest days, SCSI has obeyed a command model, which means that every device attached to a SCSI controller has a command driven state mode; however, this state model tends to differ radically by device type. This means that most Operating System's SCSI subsystem implementations tend to consist of device drivers (which understand the device command model) sitting on top of a more generic command handling mechanism which understands how to send commands to devices. This split was also reflected in the first standard for operating system interfaces to SCSI: CAM [6].

2.1 SCSI CAM

The object of CAM, as the name implies was to provide a set of common access methods that would be identical across all operating systems. Looking at figure 1 one can see how the CAM infrastructure was laid out.

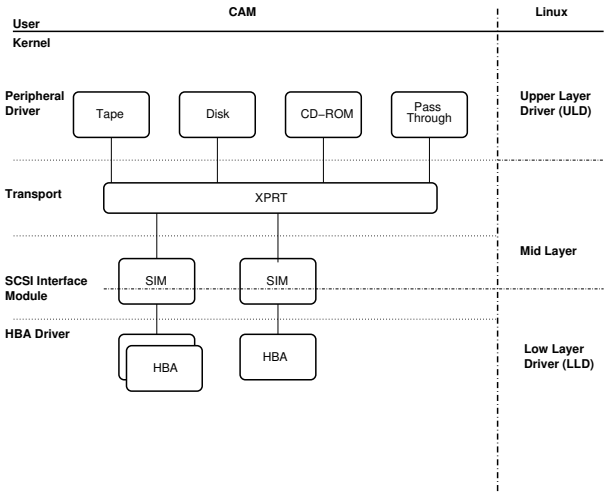

Figure 1: Illustration of CAM methods with a comparison to the current Linux SCSI subsystem

The CAM four level infrastructure on the left is shown against the current Linux three level infrastructure. The object of the comparison isn't to describe the layers in detail but to show that they map identically at the peripheral driver layer and then disconnect over the remaining ones.

Although CAM provided a good model to follow in the SCSI-2 days, it was very definitely tied to the parallel SCSI transport that SCSI-2 was based on and didn't address very well the needs of the new transport infrastructures like Fibre Channel. There was an attempt to produce a new specification taking these into account (CAM-3) but it never actually managed to produce a specification.

2.2 SCSI-3 The Next Generation

From about 1995 onwards, there was a movement to revolutionise the SCSI standard [9]. The basic thrust was a new Architecture Model (called SAM) whereby the documents were split up into Peripheral Driver command, a primary core and transport specific standards. The basic idea was to unbind SCSI from the concept of a parallel bus and make it much more extensible in terms of transport architectures.

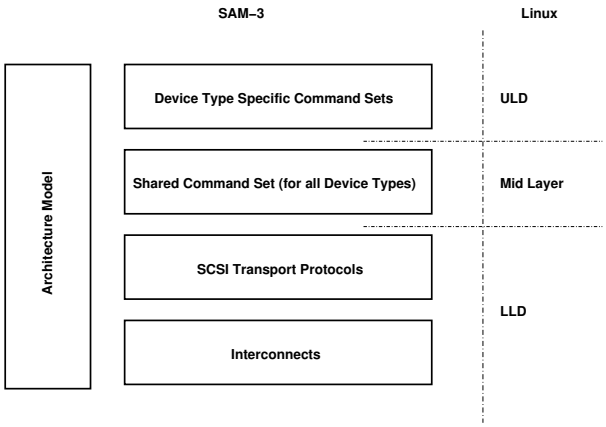

Figure 2: SAM-3 with its corresponding mapping to Linux on the right

The actual standard [8] describes the layout as depicted in figure 2 which compares almost

exactly to the layout of the Linux SCSI subsystem. Unfortunately, the picture isn't quite as rosy as this and there are certain places in the mid-layer, most notably in error handling, where we still make transport dependent assumptions.

3 Linux SCSI Subsystem

From the preceding it can be seen that the original SCSI Subsystem didn't follow either CAM or SAM exactly (although the implementation is much closer to SAM). Although the SCSI mid layer (modulo error handling) is pretty efficient now in the way it handles commands, it still lacks fine grained multi-level control of devices that CAM allows. However, in spite of this the property users most want to know about their devices (what is the maximum speed this device is communicating to the system) was lacking even from CAM.

3.1 Things Linux Learned from CAM

The basic thing CAM got right was splitting the lower layers (see figure 1) into XPRT (generic command transport) SIM (HBA specific processing) and HBA (HBA driver) was heading in the right direction. However, there were several basic faults in the design:

1. Even the XPRT which is supposed to be a generic command transport had knowledge of parallel SCSI specific parameters.
2. The User wasn't given a prescribed method for either reading or altering parameters they're interested in (like bus speed).
3. The SIM part allowed for there being one unique SIM per HBA driver.

Point 3 looks to be an advantage because it allows greater flexibility for controlling groups of HBAs according to their capabilities. However, its disadvantage is failing to prescribe precisely where the dividing line lies (i.e. since it permits one SIM per HBA, most driver writers wrote for exactly that, their own unique SIM).

A second issue for Linux is that the XPRT layer is actually split between the generic block layer and the SCSI mid-layer. Obviously, other block drivers are interested in certain features (like tags and command queueing) whereas some (like bus scanning or device identification) are clearly SCSI specific. Thus, the preferred implementation should also split the XPRT into a block generic and a SCSI specific piece, with heavy preference on keeping the SCSI specific piece as small as possible.

3.2 Recent Evolution

The policy of slimming down SCSI was first articulated at the Kernel Summit in 2002 [3] and later refined in 2003 [4]. The idea was to slim down SCSI as far as possible by moving as much of its functionality that could be held in common up to the block layer (the exemplar of this at the time being tag command queueing). and to make the mid-layer a small compact generic command processing layer with plug in helper libraries to assist the device drivers with transport and other issues. However, as is the usual course, things didn't quite go according to plan. Another infrastructure was seeping into SCSI: generic devices and `sysfs`.

3.3 `sysfs`

SCSI was the first device driver subsystem to try to embrace `sysfs` fully. This was done purely out of selfish reasons: Users were requesting extra information which we could export via `sysfs` and also, moving to the `sysfs`

infrastructure promised to greatly facilitate the Augean scale cleaning task of converting SCSI to be hotplug compliant. The way this was done was to embed a generic device into each of the SCSI device components (host, target and device) along with defining a special SCSI bus type to which the ULDs now attach as `sysfs` drivers.

However, once the initial infrastructure was in place, with extra additions that allowed drivers to export special driver specific parameters, it was noticed that certain vendor requirements were causing them to push patches into drivers that were actually exporting information that was specific to the actual transport rather than the driver [11].

Since this export of information fitted the general pattern of the "helper libraries" described above, discussion ensued about how best to achieve this export in a manner that could be utilised by all drivers acting for the given transport [12]. And thus, the concept of a Transport Class was born.

4 Transport Classes

The original concept of a transport class was that it was an entity which attached to the SCSI device at three levels (host, target and LUN) and that it exported properties from these devices straight to the user via the `sysfs` class interface. A further refinement was that the transport class (although it had support from the mid-layer) had no API that it exported to (or via) the mid layer (this is essential for allowing HBA's that aren't transport class compliant to continue to operate; however, it also has the extremely advantageous property of ensuring that the transport class services aren't bounded by any API of the mid-layer and thus makes them truly extensible). Figure 3 illustrates the relationships between transport classes and the rest of the Linux system.

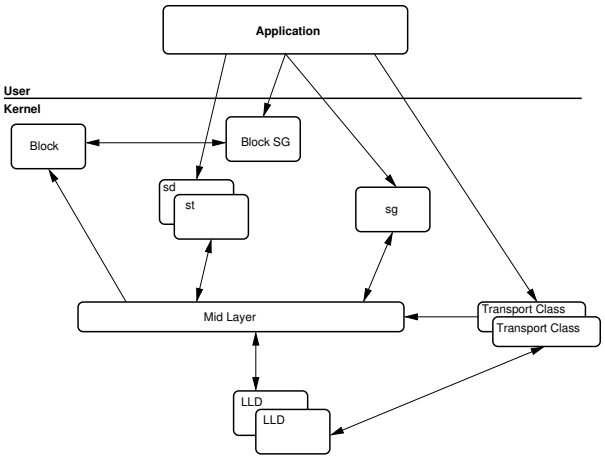

Figure 3: SCSI transport classes under Linux

4.1 Implementation

This section describes historical implementation only, so if you want to know how the classes function now[1] see section 5.3. The original implementation was designed to export transport specific parameters, so the code in the SCSI subsystem was geared around defining the class and initialising its attribute files at the correct point in the `sysfs` tree. However, once this was done, it was fairly easy to export an API from the transport class itself that could make use of these parameters (like Domain Validation for SPI, see below).

The key point was that the interaction between the mid-layer and the transport class was restricted to the mid-layer providing an API to get all the `sysfs` properties initialised and exported correctly.

[1] or rather, at the time of writing, which corresponds to the 2.6.12 kernel

4.2 Case Study: the SPI transport Class

SPI means SCSI Parallel Interface and is the new SCSI-3 terminology for the old parallel bus. In order to ascertain and control the speed of the bus, there are three essential characteristics: period, offset and width (plus a large number of minor characteristics that were added as the SPI standard evolved).

Once the ability to fetch and set these characteristics had been added, it was natural to add a domain validation [7] capability to the transport class. What domain validation (DV) does is to verify that the chosen transport characteristics match the capability of the SCSI transport by attempting to send and receive a set of prescribed patterns over the bus from the device and adjust the transport parameters if the message is garbled As the parallel bus becomes faster and faster, this sort of line clearing becomes essential since just a small kink in the cable may produce a large number of errors at the highest transfer speed.

Since the performance of Domain Validation depends on nothing more than the setting of SPI transfer parameters, it is an ideal candidate service to be performed purely within the SPI transport class. Although domain validation is most important in the high speed controllers, it is still useful to the lower speed ones. Further, certain high speed controllers themselves contain Domain Validation internally adding code bloat at best and huge potential for incorrectness at worst (the internal Domain Validation code has proved to be a significant source of bugs in certain drivers). As an illustration of the benefit, the conversion of the `aic7xxx` driver to the transport class domain validation resulted in the removal of 1,700 lines of code [2].

4.3 The Fibre Channel Transport Class

Of all the SCSI transport classes in flux at the moment, the FC class is doing the most to revolutionise the way the operating system sees the transport. Following a fairly huge program of modification, the FC transport class is able to make use of expanded mid-layer interfaces to cause even non-SCSI ports of the fabric to appear under the SCSI device tree—even the usual SCSI device structure is perturbed since the tree now appears as host/rport/target/device.

The object of this transport class is twofold:

1. Consolidate all services for Fibre Channel devices which can be held in common (things like cable pull timers, port scanning), thus slimming down the actual Fibre Channel drivers.

2. Implement a consistent API via `sysfs` which all drivers make use of, thus in theory meaning a single SAN management tool can be used regardless of underlying HBA hardware.

5 Transport Class Evolution

Looking at what's happening in the SCSI world today, it's clear that the next nascent transport to hit the Linux Kernel will be the Serial Attached SCSI (SAS) one. Its cousin, Serial ATA (SATA) is already present in both the 2.4 and 2.6 kernels. One of the interesting points about SAS and SATA is that at the lowest level, they both share the same bus and packet transport mechanism (the PHY layer, which basically represent a physical point to point connection which may be only part of a broader logical point to point connection).

The clear direction here is that SAS should have *two* separate transport classes: one for SAS itself and one for the PHY, and further that the PHY transport class (which would control the physical characteristics of the PHY interface) should be common between SAS and SATA.

5.1 Multiple Transport Classes per Device

In the old transport class paradigm, each transport class requires an "anchor" in the enveloping device structure (for SCSI we put these into `struct Scsi_Host`, `struct scsi_target`, and `struct scsi_device`). However, to attach multiple transport classes under this paradigm, we'd have to have multiple such anchors in the enveloping device which is starting to look rather inefficient.

The basic anchor that is required is a pointer to the class and also a list of attributes which appear as files in `sysfs`, so the solution is to remove the need for this anchor altogether: the generic attribute container.

5.2 Generic Attribute Containers

The idea here is to dispense entirely with the necessity for an anchor within some enveloping structure. Instead, all the necessary components and attribute files are allocated separately and then matched up to the corresponding generic device (which currently always sits inside the enveloping structure). The mechanism by which attribute containers operate is firstly by the pre-registration of a structure that contains three elements:

1. A pointer to the class,
2. a pointer to the set of class device attributes
3. and a match callback which may be coded to use subsystem specific knowledge to determine if a given generic device should have the class associated with it.

Once this is registered, a set of event triggers on the generic device must be coded into the subsystem (of necessity, some of these triggers are device creation and destruction, which are used to add and remove the container, but additional triggers of any type whatever may also be included). The benefit of these triggers is enormous: the trigger function will be called for all devices to whom the given class is registered, so this can be used, for instance, to begin device configuration. Once the generic attribute container was in place, it was extremely simple to build a generic transport class on top of it.

5.3 Generic Transport Classes

Looking at the old SCSI transport classes in the light of the new attribute containers, it was easily seen that there are five trigger points:

1. setup (mandatory), where the class device is created but not yet made visible to the system.
2. add (mandatory), where the created class device and its associated attributes are now made visible in `sysfs`
3. configure (optional), which is possibly more SCSI-centric; the above two operations (setup and add) probe the device using the lowest common transport settings. Configure means that the device has been found and identified and is now ready to be brought up to its maximum capabilities.
4. remove (mandatory), where the class device should be removed from the `sysfs` export preparatory to being destroyed.

5. destroy (mandatory), called on final last put of the device to cause the attribute container to be deallocated.

All of these apart from configure are essentially standard events that all generic devices go through. Basically then, a generic transport class is a structure containing three of the five trigger points (add, configure and remove; setup and destroy being purely internally concerned with allocation and deallocation of the transport class, with no external callback visibility). To make use of the generic transport container, all the subsystem has to do is to register the structure with the three callbacks (which is usually done in the transport class initialisation routine) and embed the mandatory trigger points into the subsystem structure creation routines as `transport_`*`event`*`_device()`.

As a demonstration of the utility of the generic transport class, the entire SCSI transport infrastructure was converted over to the generic transport class code with no loss of functionality and a significant reduction in lines of code and virtually no alteration (except for initialisations) within the three existing SCSI transport classes.

Finally, because the generic transport class is built upon the generic attribute containers, which depend only on the `sysfs` generic device, any subsystem or driver which has been converted to use generic devices may also make use of generic transport classes.

6 So Where Are We Going?

Although the creation of the generic transport classes was done for fairly selfish reasons (to get SAS to fit correctly in the transport framework with two attached classes), the potential utility of a generic transport infrastructure extends well beyond SCSI.

6.1 IDE and `hdparm`

As the ATA standards have evolved [10], the transport speed and feature support (like Tag Command Queueing) has also been evolving.

Additionally, with the addition of SATA and AoE (ATA over Ethernet), IDE is evolving in the same direction that SCSI did many years ago (acquiring additional transports), so it begins to make sense to regroup the currently monolithic IDE subsystem around a core command subsystem which interacts with multiple transports.

Currently if you want to see what the transfer settings of your drive are, you use the `hdparm` program, which manipulates those settings via special ioctls. This same information would be an ideal candidate for exporting through `sysfs` via the generic transport classes.

6.2 Hardware RAID

The kernel today has quite a plethora of hardware RAID drivers; some, like `cciss` are present in the block subsystem but the majority are actually presented to the system as SCSI devices. Almost every one of these has a slew of special ioctls for configuration, maintenance and monitoring of the arrays, and almost all of them comes with their own special packages to interface to these private ioctls. There has recently been a movement in the standards committees to unify the management approach (and even the data format) of RAID arrays, so it would appear that the time is becoming ripe for constructing a raid management transport class that would act as the interface between a generic management tool and all of the hardware RAID drivers.

6.3 SAS

As has been mentioned before, the need to have both a SAS and a PHY class for the same device was one of the driving reasons for the creation of the generic transport class. We are also hoping that SAS will be the first SCSI transport to enter the kernel with a fully fledged transport class system (both SPI and FC had their transport classes grafted on to them after drivers for each had been accepted into the kernel, and not all FC or SPI drivers currently make use of the capabilities afforded by the transport classes).

Hopefully, the vastly improved functionality provided to FC drivers by the FC transport class, with the addition of the concept of the remote port and transport class driven domain enumeration will at least have convinced the major SAS protagonists of the benefits of the approach. However, the current statement of the SCSI maintainers has been that a working SAS transport class is a necessary prerequisite for inclusion of any SAS driver.

6.4 SCSI Error Handling

One of the last major (and incredibly necessary) re-organisations of SCSI involves cleaning up the error handler. Currently, the SCSI error handler is completely monolithic (i.e. it applies to every driver) and its philosophy of operation is still deeply rooted in the old parallel bus, which makes it pretty inappropriate for a large number of modern transports. Clearly, the error handler should be transport specific, and thus it would make a natural candidate for being in a transport class. However, previously transport classes took services from the mid-layer but didn't provide any services to it (the provide services only to the LLD). However, an error handler primarily provides services to the Mid Layer and an API for handling errors to the LLD, so it doesn't quite fit in with the original vision for the SCSI transport classes. However, it does seem that it can be made to conform more closely with the generic transport class, where the error handler classes become separate from the actual "transport" transport classes.

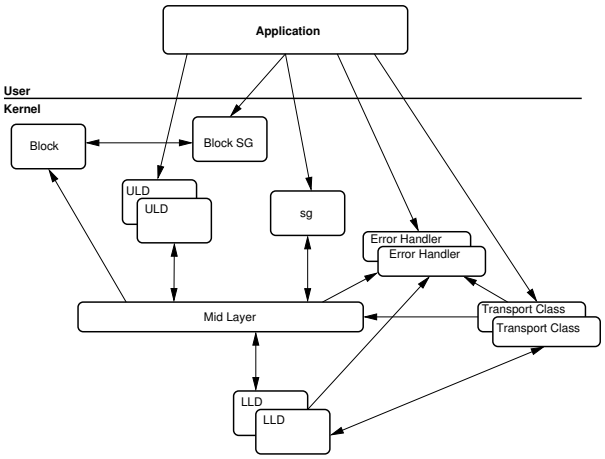

Figure 4: An illustration of how the error handlers would work as generic transport classes

How this would work is illustrated in figure 4 The arrows represent the concept of "uses the services of." The idea essentially is that the error handler classes would be built on the generic transport classes but would provide a service to the mid-layer based on a transport dependent API. The error handler parameters would, by virtue of the `sysfs` component, be accessible to the user to tweak.

7 Conclusions

The SCSI transport classes began life as helper libraries to slim down the SCSI subsystem. However, they subsequently became well defined transport class entities and went on to spawn generic transport classes which have utility far beyond the scope of the original requirement.

Two basic things remain to be done, though:

1. Retool SCSI error handling to be modular using generic transport classes.
2. Actually persuade someone outside of the SCSI subsystem to make use of them.

References

[1] Secretariat: Computer & Business Equipment Manufacturers Association. Small computer system interface - 2, 1993. http://www.t10.org/ftp/t10/drafts/s2/s2-r10l.pdf.

[2] James Bottomley. [PATCH] convert aic7xxx to the generic Domain Validation, April 2005. http://marc.theaimsgroup.com/?l=linux-scsi&m=111325605403216.

[3] James E.J. Bottomley. Fixing SCSI. USENIX Kernel Summit, July 2002. http://www.steeleye.com/support/papers/scsi_kernel_summit.pdf.

[4] James E.J. Bottomley. SCSI. USENIX Kernel Summit, July 2003. http://www.steeleye.com/support/papers/scsi_kernel_summit_2003.pdf.

[5] Secretariat: Information Technology Industry Council. SCSI-3 architecture model, 1995. http://www.t10.org/ftp/t10/drafts/sam/sam-r18.pdf.

[6] ASC X3T10 Technical Editor. SCSI-2 common access method transport and scsi interface module, 1995. http://www.t10.org/ftp/t10/drafts/cam/cam-r12b.pdf.

[7] NCITS T10 SDV Technical Editor. SCSI domain validation (SDV), 2001. http://www.t10.org/ftp/t10/drafts/sdv/sdv-r08b.pdf.

[8] T10 Technical Editor. SCSI architecture model - 3, 2004. http://www.t10.org/ftp/t10/drafts/sam3/sam3r14.pdf.

[9] T10 Technical Editors. Collection of SCSI-3 protocol specifications. There are too many to list individually, but they can all be found at: http://www.t10.org/drafts.htm.

[10] T13 Technical Editors. Collection of ATA protocol specifications. There are too many to list individually, but they can all be found at: http://www.t13.org.

[11] Martin Hicks. [PATCH] Transport Attributes Export API, December 1993. http://marc.theaimsgroup.com/?l=linux-scsi&m=107289789102940.

[12] Martin Hicks. Transport attributes – attempt#4, January 1994. http://marc.theaimsgroup.com/?l=linux-scsi&m=107463606609790.